REMARKS

The Examiner has rejected Claims 1, 2, 4-14, 16-26, and 28-42 under 35 U.S.C. 102(e) as being anticipated by Haatainen et al. (U.S. Patent No. 6,678,734 B1). Applicant respectfully disagrees with such rejection.

With respect to the independent claims, the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique of "intercepting a portion of outgoing network data characteristic of the operating system" (see this or similar, but not necessarily identical language in the independent claims).

> "The firewall hooking approach is only available for a very limited range of operating systems, such as Linux and FreeBSD. Furthermore, on many operating systems the functionality may not be enabled by default. Thus, using the firewall hooking features may require the user to recompile and install a new operating system kernel, which exceeds the skills of most system administrators. Therefore the known firewall hooking solutions are not generally usable and do not solve the problem of providing packet intercepting for the most popular operating systems or for large numbers of unskilled users.
>
> Specialized applications such as anti-virus software have trapped system services to monitor e.g. file system activity. Some operating systems provide special support for such applications. For example, the Windows 95 operating system provides a VMM. VxD. Hook_Device_Service function for hooking device services for such applications, as described for example in the reference marked as Win95DDK in the enclosed list of references.
>
> Another known method for inspecting network traffic is to replace the known WINSOCK.DLL file on Windows. This file does not see individual data packets, but instead gets access to each Windows SOCKET on the system. It is possible to move the original file aside and replaced it by a new DLL so that the new DLL calls the original one. In known systems, the replacement has been performed at installation time, and the replacement module has been loaded when an application using it starts (not, e.g., when the system boots). Although conceptually somewhat related to hooking, this method is not directly applicable to packet interception because access to packets is not obtained.
>
> The very basic idea of using hooking to get access to network packets is known, as described for example in the references marked as Lanciani98, Lanciani98PPPMAC and Lanciani98Reply in the enclosed list of references. It has been used in certain protocol

> conversion applications. The known prior art in this area consists of a program that implements the packet driver programming interface on top of the NDIS programming interface, as described for example in the reference marked as Lanciani97NDIS3PKT in the enclosed list of references. Other known alternatives are the implementation of an ODI packet driver over NDIS (reference Lanciani96ODIPKT3.1), an older packet driver interface (reference Lanciani92DIS_PKT), a Banyan Vines packet driver (reference Lanciani94IBANPKT), and a NFS client for Windows (reference Lanciani97NFSTDI). However, a protocol conversion or an alternative packet driver can not support sophisticated network security applications; they can even complicate the use of security arrangements since they may cause unexpected and disallowed changes in the form of the packets." (Col. 7, lines 11-60 - emphasis added)

Applicant respectfully asserts that the excerpt from the above reference relied upon by the Examiner merely teaches that "the known firewall hooking solutions are not generally usable and do not solve the problem of providing packet intercepting for the most popular operating systems" (emphasis added). Haatainen continues to teach that "[t]he very basic idea of using hooking to get access to network packets is known" (emphasis added) and has "...been used in certain protocol conversion applications" (emphasis added). However, merely suggesting "hooking to get access to network packets" simply fails to disclose a technique of "intercepting a portion of outgoing network data characteristic of the operating system" (emphasis added), as claimed by applicant.

Further, with respect to the independent claims, the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique of "conditionally masking the portion of outgoing network data to impersonate a different operating system in accordance with a security policy if the network is an untrusted network" (see this or similar, but not necessarily identical language in the independent claims).

> "The firewall hooking approach is only available for a very limited range of operating systems, such as Linux and FreeBSD. Furthermore, on many operating systems the functionality may not be enabled by default. Thus, using the firewall hooking features may require the user to recompile and install a new operating system kernel, which exceeds the skills of most system administrators. Therefore the known firewall hooking solutions

are not generally usable and do not solve the problem of providing packet intercepting for the most popular operating systems or for large numbers of unskilled users.

Specialized applications such as anti-virus software have trapped system services to monitor e.g. file system activity. Some operating systems provide special support for such applications. For example, the Windows 95 operating system provides a VMM. VxD. Hook_Device_Service function for hooking device services for such applications, as described for example in the reference marked as Win95DDK in the enclosed list of references.

Another known method for inspecting network traffic is to replace the known WINSOCK.DLL file on Windows. This file does not see individual data packets, but instead gets access to each Windows SOCKET on the system. It is possible to move the original file aside and replaced it by a new DLL so that the new DLL calls the original one. In known systems, the replacement has been performed at installation time, and the replacement module has been loaded when an application using it starts (not, e.g., when the system boots). Although conceptually somewhat related to hooking, this method is not directly applicable to packet interception because access to packets is not obtained.

The very basic idea of using hooking to get access to network packets is known, as described for example in the references marked as Lanciani98, Lanciani98PPPMAC and Lanciani98Reply in the enclosed list of references. **It has been used in certain protocol conversion applications.** The known prior art in this area consists of a program that implements the packet driver programming interface on top of the NDIS programming interface, as described for example in the reference marked as Lanciani97NDIS3PKT in the enclosed list of references. Other known alternatives are the implementation of an ODI packet driver over NDIS (reference Lanciani96ODIPKT3.1), an older packet driver interface (reference Lanciani92DIS_PKT), a Banyan Vines packet driver (reference Lanciani94IBANPKT), and a NFS client for Windows (reference Lanciani97NFSTDI). **However, a protocol conversion or an alternative packet driver can not support sophisticated network security applications; they can even complicate the use of security arrangements since they may cause unexpected and disallowed changes in the form of the packets."** (Col. 7, lines 11-60 - emphasis added)

Applicant respectfully asserts that the above excerpt relied upon by the Examiner suggests that "[t]he very basic idea of using hooking to get access to network packets is known" (emphasis added) and "has been used in certain protocol conversion applications" (emphasis added). Haatainen continues to suggest that "a protocol conversion or an alternative packet driver ... can even complicate the use of security arrangements since they may cause unexpected and disallowed changes in the form of the packets" (emphasis added).

Simply disclosing the use of "hooking to get access to network packets," however, completely fails to even suggest "conditionally masking the portion of outgoing network data to impersonate a different operating system" (emphasis added). In addition, there is no disclosure in the referenced excerpt of a technique for "conditionally masking the portion of outgoing network data to impersonate a different operating system in accordance with a security policy if the network is an untrusted network" (emphasis added), as claimed by applicant.

In addition, with respect to the independent claims, the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique "wherein masking the portion comprises: replacing the portion of outgoing network data with data characteristic of the different operating system" (see this or similar, but not necessarily identical language in the independent claims).

> "The firewall hooking approach is only available for a very limited range of operating systems, such as Linux and FreeBSD. Furthermore, on many operating systems the functionality may not be enabled by default. Thus, using the firewall hooking features may require the user to recompile and install a new operating system kernel, which exceeds the skills of most system administrators. Therefore the known firewall hooking solutions are not generally usable and do not solve the problem of providing packet intercepting for the most popular operating systems or for large numbers of unskilled users.
>
> Specialized applications such as anti-virus software have trapped system services to monitor e.g. file system activity. Some operating systems provide special support for such applications. For example, the Windows 95 operating system provides a VMM. VxD. Hook_Device_Service function for hooking device services for such applications, as described for example in the reference marked as Win95DDK in the enclosed list of references.
>
> Another known method for inspecting network traffic is to replace the known WINSOCK.DLL file on Windows. This file does not see individual data packets, but instead gets access to each Windows SOCKET on the system. It is possible to move the original file aside and replaced it by a new DLL so that the new DLL calls the original one. In known systems, the replacement has been performed at installation time, and the replacement module has been loaded when an application using it starts (not, e.g., when the system boots). Although conceptually somewhat related to

hooking, this method is not directly applicable to packet interception because access to packets is not obtained.

The very basic idea of using hooking to get access to network packets is known, as described for example in the references marked as Lanciani98, Lanciani98PPPMAC and Lanciani98Reply in the enclosed list of references. It has been used in certain protocol conversion applications. The known prior art in this area consists of a program that implements the packet driver programming interface on top of the NDIS programming interface, as described for example in the reference marked as Lanciani97NDIS3PKT in the enclosed list of references. Other known alternatives are the implementation of an ODI packet driver over NDIS (reference Lanciani96ODIPKT3.1), an older packet driver interface (reference Lanciani92DIS_PKT), a Banyan Vines packet driver (reference Lanciani94IBANPKT), and a NFS client for Windows (reference Lanciani97NFSTDI). However, a protocol conversion or an alternative packet driver can not support sophisticated network security applications; they can even complicate the use of security arrangements since they may cause unexpected and disallowed changes in the form of the packets." (Col. 7, lines 11-60 - emphasis added)

Applicant respectfully asserts that the excerpt relied upon by the Examiner simply discloses "using hooking to get access to network packets" (emphasis added) for use in "certain protocol conversion applications." Haatainen continues to disclose that "a protocol conversion or an alternative packet driver ... can even complicate the use of security arrangements since they may cause unexpected and disallowed changes in the form of the packets" (emphasis added). However, causing "unexpected and disallowed changes in the form of the packets" simply fails to even suggest a technique "wherein masking the portion comprises: replacing the portion of outgoing network data with data characteristic of the different operating system" (emphasis added), as claimed by applicant.

The Examiner is reminded that a claim is anticipated only if each and every element as set forth in the claim is found, either expressly or inherently described in a single prior art reference. *Verdegaal Bros. v. Union Oil Co. Of California*, 814 F.2d 628, 631, 2 USPQ2d 1051, 1053 (Fed. Cir. 1987). Moreover, the identical invention must be shown in as complete detail as contained in the claim. *Richardson v. Suzuki Motor Co.*868 F.2d 1226, 1236, 9USPQ2d 1913, 1920 (Fed. Cir. 1989). The elements must be arranged as required by the claim.

This criterion has simply not been met by the above reference, as noted above. A notice of allowance or specific prior art showing of each of the foregoing claim elements, in combination with the remaining claimed features, is respectfully requested.

Applicant further notes that the prior art is also deficient with respect to the dependent claims. For example, with respect to Claim 4 et al., the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique "wherein the security policy identifies the portion of outgoing network data and specifies an action to take to mask the portion of outgoing network data" (see this or similar, but not necessarily identical language in the dependent claims).

> "Windows NT does not provide a convenient interface for hooking DLL entry points like the one available for Windows 95 and 98 VxDs. It would be possible to hook the NDIS functions by replacing code from the beginning of the original functions; however, this is rather complicated. An alternative but conceptually similar approach is described here.
>
> Basically, one can achieve functionality similar to hooking by moving aside (saving) the original operating system module (NDIS.SYS), replacing it by a new module that implements the same programming interface, performs interceptions, and calls the original operating system module whenever it wishes to perform the original operation." (Col. 15, lines 53-65 - emphasis added)

Applicant respectfully asserts that the excerpt from the above reference relied upon by the Examiner teaches that "one can achieve functionality similar to hooking by moving aside (saving) the original operating system module (NDIS.SYS), replacing it by a new module that implements the same programming interface, performs interceptions, and calls the original operating system module whenever it wishes to perform the original operation" (emphasis added). However, using a replacement module to perform interceptions simply fails to disclose the use of a "security policy," as claimed by applicant. Further the disclosure of "perform[ing] interceptions" (emphasis added) fails to even suggest a technique "wherein the security policy identifies the portion of outgoing network data and specifies an action to take to mask the portion of outgoing network data" (emphasis added), as claimed by applicant.

Further, with respect to Claim 5 et al., the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique "wherein the security policy further specifies replacement data for the portion of outgoing network data, the replacement data characteristic of the different operating system" (see this or similar, but not necessarily identical language in the dependent claims).

> 'One possible solution to this problem is to have a separate program run after each reboot, and have that program check if the module has changed, and if so, replace it with the interceptor module.
>
> A second solution is to create what is called a native application: an application that runs before the operating system loads the networking modules. A native application runs over the NT kernel but without Win32 or other user mode subsystems. A native application is thus a user mode application running over NT, but not really a "Windows" application. Such applications are traditionally used e.g. for boot-time cleanups, such as file system consistency and integrity checks. A method that solves all three problems above works as follows:
>
> At boot, run a native program. This program first checks if the kernel module (e.g. NDIS.SYS) is already a replacement generated by itself If so, it removes the replacement and restores the original. Then, it begins its real work. It moves the original aside, reads its entry point table, reads the entry point table of the interceptor module, and dynamically creates a new module that contains the same entry points as the original module, and forwards each entry point to the interceptor module if that entry point exists in the interceptor module, and to the original module if the entry point does not exist in the interceptor module. The generated module is written in the place of the original module.
>
> Boot then continues, and the generated module is loaded. The original module and the interceptor module are also loaded.
>
> When the operating system has been loaded, a program (e.g., system service application) is run that removes the generated DLL and instead moves the original operating system module (NDIS.SYS) back.
>
> This method provides a high level of robustness against differences between service packs, as it is quite rare that interfaces would change in completely incompatible ways (as opposed to adding new interfaces). It also avoids the problems in installing service packs, as the original kernel module will

> always be there when a service pack is installed, and the new module will get used after the service pack has been installed.
>
> It should be noted that the above procedure could be implemented in multiple ways. The operations can be performed in different orders, other interleaving operations may be added, and some of the operations left out without substantially affecting the ideas behind the procedure.' (Col. 16, lines 13-59 - emphasis added)

Applicant respectfully asserts that the excerpt from Haatainen relied upon by the Examiner merely teaches a method of creating and loading an "interceptor module." Specifically, the excerpts disclose "dynamically creat[ing] a new module that contains the same entry points as the original module, and forward[ing] each entry point to the interceptor module if that entry point exists in the interceptor module, and to the original module if the entry point does not exist in the interceptor module" (emphasis added). However, there simply is no disclosure in the excerpt above of a technique "wherein the security policy further specifies replacement data for the portion of outgoing network data, the replacement data characteristic of the different operating system" (emphasis added), as claimed by applicant.

In addition, with respect to Claim 9 et al., the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique of "transmitting the portion of outgoing network data unchanged if the network is a trusted network" (see this or similar, but not necessarily identical language in the dependent claims).

> "To receive incoming packets from the adapters 1204, the replacement input (ip_input) function 1205 within the packet interceptor gets called. To send incoming packets to protocols 1201, the interceptor calls the original input (ip_input) function 1206.
>
> In the second embodiment of the invention the interceptor gets information about active network interfaces by looking at the ifnet global variable or, in other similarly acting operating system, a corresponding other variable in the kernel. This variable contains a list of network interfaces, and each interface data structure contains the protocol addresses of the interface.
>
> Further in the second embodiment of the invention the interceptor learns about new network interfaces (e.g. dial-up lines going up

> or down) when the replacement ifioctl function gets called. After calling the original ifioctl function the interceptor can then reread the list of network interfaces, and learn of any changes that way." (Col. 14, lines 6-23 - emphasis added)

Applicant respectfully asserts that the excerpt above relied upon by the Examiner merely teaches that "the interceptor gets information about active network interfaces by looking at the ifnet global variable or, in other similarly acting operating system, a corresponding other variable in the kernel" (emphasis added). Haatainen continues to teach that "[t]his variable contains a list of network interfaces, and each interface data structure contains the protocol addresses of the interface" (emphasis added). However, merely retrieving "a list of network interfaces" and "the protocol addresses of the interface" simply fails to disclose "if the network is a trusted network" (emphasis added), in the context claimed by applicant. Further, the disclosure that in order to "send incoming packets to protocols 1201, the interceptor calls the original input (ip_input) function 1206" (emphasis added) simply fails to disclose a technique of "transmitting the portion of outgoing network data unchanged if the network is a trusted network" (emphasis added), as claimed by applicant.

Moreover, with respect to Claim 42, the Examiner has relied on the following excerpt from the above reference to make a prior art showing of applicant's claimed technique "wherein the false response is sent if the operating system would normally not respond to the incoming network data" (see this or similar, but not necessarily identical language in the dependent claims).

> "If the unloading of the kernel module according to the second embodiment of the invention is initiated as at step 1301 of FIG. 13, it is most advantageous to disable interrupts and restore the functions to their originals as at step 1302, re-enable interrupts and possibly flush instruction cache as at step 1303, and continue with the unload as at step 1304 by re-emptying the memory locations previously allocated to the packet interceptor.
>
> In some applications, the interceptor may be interested in finding only specific adapters or protocols out of all adapters or protocols installed in the system. Thus, the interceptor may not want to determine information about all adapters. Instead, it may stop looking at such information e.g. after it has found all the network adapters it was looking for.

> In several places in the above description reference was made to hooking or replacing a function. In the appended claims the term hooking will be used collectively for such operations. There are many ways of implementing hooking." (Col. 14, lines 24-41 - emphasis added)

Applicant respectfully asserts that the excerpt above relied upon by the Examiner merely teaches a method of "unloading ... the kernel module" (emphasis added). Haatainen discloses that "it is most advantageous to disable interrupts and restore the functions to their originals as at step 1302, re-enable interrupts and possibly flush instruction cache as at step 1303..." (emphasis added). However, disabling and re-enabling interrupts simply fails to suggest a technique "wherein the false response is sent if the operating system would normally not respond to the incoming network data" (emphasis added), as claimed by applicant.

Again, the foregoing anticipation criterion has simply not been met by the above reference, as noted above. A notice of allowance or specific prior art showing of each of the foregoing claim elements, in combination with the remaining claimed features, is respectfully requested.

Thus, all of the independent claims are deemed allowable. Moreover, the remaining dependent claims are further deemed allowable, in view of their dependence on such independent claims.

In the event a telephone conversation would expedite the prosecution of this application, the Examiner may reach the undersigned at (408) 505-5100. The Commissioner is authorized to charge any additional fees or credit any overpayment to Deposit Account No. 50-1351 (Order No. NAI1P350).

Respectfully submitted,
Zilka-Kotab, PC.

Kevin J. Zilka
Registration No. 41,429

P.O. Box 721120
San Jose, CA 95172-1120
408-505-5100